

IN THE SPECIFICATION

Please amend the specification as follows:

The paragraph beginning at page 2, line 17, is amended as follows:

The channel adapter receives a virtual address or a portion of a virtual address from the operating system or application and converts this virtual address into a physical address of a memory location in system memory. Moreover, a channel adapter monitors progress of tasks or [[work]] ongoing work to determine when the work has been completed, whereby the channel adapter may store a completion status so that the application or operating system may be alerted as to the completion of the work.

The paragraph beginning at page 4, line 12, is amended as follows:

FIG. 4 is a schematic diagram of details of a request register according to an example embodiment of the present invention;

The paragraph beginning at page 6, line 8, is amended as follows:

Further, arrangements may be shown in block diagram form in order to avoid obscuring the invention, and also in view of the fact that specifics with respect to implementation of such block diagram arrangements is highly dependent upon the platform within which the present invention is to be implemented, i.e., specifics should be well within the purview of one skilled in the art. Where specific details (e.g., circuits, flowcharts) are set forth in order to describe example embodiments of the invention, it should be apparent to one skilled in the art that the invention can be practiced without these specific details. Finally, it should be apparent that any combination of hard-wired circuitry and software instructions can be used to implement embodiments of the present invention, i.e., the present invention is not limited to any specific combination of hardware circuitry and software instructions.

The paragraph beginning at page 7, line 6, is amended as follows:

The present invention relates to apparatus and method for enhanced channel adapter performance through implementation of an [[a]] address translation engine and completion queue engine. The address translation engine supports a two level translation and protection table (TPT). The translation and protection table is a construct resident in system memory which may be used by a channel adapter to convert virtual addresses (e.g., used by software, such as operating systems, applications, etc.) into physical addresses, used by the channel adapter. The completion queue engine, similar to the address translation engine, operates independently of the packet processing function, and manages completion queue and event queue processing.

The paragraph beginning at page 12, line 15, is amended as follows:

FIG. 2 shows a block diagram of an address translation engine according to an example embodiment of the present invention. Address translation engine 12 consists of two major subblocks, inbound request logic 32 and request completion logic 34. Address translation engine 12 may communicate with portions of channel adapter 10 requiring address translation requests. These portions may include packet processing engine 16 and completion queue engine 14 (FIG. 1). Requests for address translation may come from sources external to channel adapter 10 and still be within the spirit and scope of the present invention. Address translation engine inbound request logic 32 supports a set of request registers used to buffer requests for address translations. Valid physical addresses may be returned to a requestor via request completion logic 34. Address translation engine 12 also communicates with host interface 18 in order to access system memory. Requests for system memory reads may be issued by inbound request logic 32 and are returned and processed by request completion logic 34. Request completion logic 34 supports a set of TPT data registers used to buffer read completions that may return back to back from host interface 18. Address translation engine 12 also supports a local bus 22. Local bus 22 provides a programming interface for registers supported by the channel adapter. Specifically, address translation engine 12 supports registers indicating the size and base physical address of TPT 26 in system memory.

The paragraph beginning at page 13, line 11, is amended as follows:

The architecture of address translation engine 12 takes advantage of the parallelism between the inbound and outbound address translation requests. Each portion of the address translation engine 12 acts as a large pipe. Requests ~~enter~~ entered into inbound request logic 32 and are passed to host interface 18. After a period of time, completions return from host interface 18 and are processed in request completion logic 34 and returned to the requester, or reintroduced to inbound request logic 32 to support a second translation request if required. The present invention allows both pipes to work largely independently of each other, only sharing an interface to handle second translation requests and error cases. Data registered with a request may also pass between pipes to allow for calculations and permission checking in request completion logic 34.

The paragraph beginning at page 13, line 22, is amended as follows:

FIG. 3 shows a diagram of inbound request logic according to an example embodiment of the present invention. Translation requests 30 may be received from a requestor (e.g., packet processing engine 16, completion queue engine 14, etc. (FIG. 1)), and passed to one or more switching devices 40. Switching device 40 switches between requests that are original requests coming from interface 30 and second translation requests that may be coming from request completion logic 34 (FIG. 2). A switching device may exist for each set of request registers 42. In this example embodiment, address translation engine 12 (FIG. 1) supports five outstanding address translation requests, (e.g., four from packet processing engine 16 and one from completion queue engine 14).

The paragraph beginning at page 14, line 8, is amended as follows:

Five sets of request registers 42 support the five outstanding address translation requests. Request register sets 42 may be used to store the information relevant for a given request. This may include the key and virtual address for the first translation for a given data buffer or just a protection index for requests in the middle of a data buffer or accessing a completion queue or event queue. All request types may require protection information. The output of each request

register set 42 may be sent to request completion logic 34 (FIG. 2) for data comparison, and sent to another switching device 46. Arbitration logic 44 performs a priority scheme (e.g., round robin) to determine which request should be processed. Once this selection is made, arbitration logic 44 controls switching device 46 to pass the appropriate protection index or key through switching device 46 and towards further processing. Arbitration logic 44 may then also clear the request register of request register set 42 for the associated request. The data register of request register set 42 may contain a variety of information, for example, a virtual address, a key, a protection index, permissions for the page, a protection domain, etc. The type register of request register set 42 denotes whether the request is a first request or a second request (i.e. from request completion logic 34 (FIG. 2)).

The paragraph beginning at page 15, line 1, is amended as follows:

FIG. 4 shows a schematic diagram of details of a single request register shown in FIG. 3 according to an example embodiment of the present invention. A requesting interface has write access to a request register. A request may be issued to address translation engine 12 (FIG. 1) by asserting a write signal and providing valid data information and an address field used to identify which piece of information is provided on the data bus. Each request may be allowed a single outstanding address translation. Request completion logic 34 may also load a request into the inbound request registers 42. This may be done to allow for the second address translation to complete autonomously without involving the requestor at all. Given that each requestor may only be allowed a single outstanding request, request completion logic 34 may overwrite the appropriate values in the original request (e.g., changing the key originally provided with the request into the calculated protection index generated to access the translation entry and modifying the type indication as appropriate). The implementation of request registers according to the present invention are advantageous in that a requestor may set up a request and then proceed to do other valid operations. For example, packet processing engine 16 may initiate a request, and then may switch to another task directly related to moving messages across the switched fabric.

The paragraph beginning at page 15, line 18, is amended as follows:

Decoder 60 decodes the address information to determine what type of data is on the data lines. Decoder 60 then generates a request and forwards the request and data to switching device 62. Switching device 62 selects between translation requests that are a first request coming from translation request 30 or a translation request (i.e., second request) coming from request completion logic 34. The request is stored in register 64 and the data in register 66 before being sent to inbound request logic 32.

The paragraph beginning at page 16, line 1, is amended as follows:

Returning to FIG. 3, once an address translation request has been stored in the request registers, a request signal may be passed to arbitration logic 44 which selects fairly between all outstanding requests and passes a single requested key or protection index to inbound request logic 32. Arbitration logic 44 associates a tag value with each request (e.g., three bits). This tag value may be used to reorder requests after being issued to host interface 18. The tag value is important in that it allows both halves of address translation engine 12 to operate independently of each other ~~from one another~~. Moreover, the tag value may be used to select the appropriate data for comparison in request logic 34.

The paragraph beginning at page 17, line 5, is amended as follows:

FIG. 5 shows a diagram of TPT base register update logic according to an example embodiment of the present invention. As seen in FIG. 5, software (e.g., an operating system or an application), can send information to upgrade the TPT base registers 52 and 54 via local bus 22. Decoder 70 determines whether the data is meant for the lower portion of the address or the high portion of the address. Generally, the TPT base low address may be sent first whereby this data is stored in temporary register 72. When the remainder of the TPT base address, i.e., high portion, arrives, this data is loaded into TPT base address register 52 concurrently with the TPT address low stored in temporary register 72 being loaded into TPT base address register 54. Therefore, both the high and low portions of the TPT base address are presented in parallel together to inbound request logic 32. Therefore, according to the present invention, software is allowed to

autonomously and independently update the base of TPT 26. If the lower address and the upper address of the TPT base are each 32 bits, the present invention allows the operating system to issue 64 bit cycles that are received and handled by channel adapter 10 in the form of two 32 bit accesses through the internal local bus 22. This allows for autonomous update of the TPT base registers 52, 54 without any knowledge of what activity is currently in progress within address translation engine 12 or channel adapter 10. Similarly, this may apply to an operating system issuing 32 bit operations to the channel adapter.

The paragraph beginning at page 18, line 1, is amended as follows:

Referring back to FIG. 3, after generation of the correct physical address based on the key or protection index value, inbound request logic 32 issues an inbound request to host interface 18. Address translation engine 12 may perform system memory operations of either 8 bytes (for a protection index request) or 24 bytes (for a key request). Reading 24 bytes, in the case of the key request, is advantageous in that should the key reference a window entry, the 24 bytes read from the TPT 26 will contain the entire window entry, including all of the protection information and the upper and lower byte address specifying the memory window bound for remote access. If the key references a region entry, then the 24 byte read may contain the protection information for that memory region as well as the first two TPT entries of that region. Should the second read of TPT 26 indicate an access to either the first or second entry in a given memory region, then address translation engine 12 may use the information for those TPT entries returned with the first 24 byte read of TPT 26. This effectively saves a second access to system memory in the case where a region entry is defined and the access begins within the first two virtual pages of that memory region. Therefore, an additional read of system memory is prevented resulting in an important performance benefit. The second read of TPT 26 is not needed since the information that would normally be read was read as a "freebie" during the first access of TPT 26. Although at first glance this may appear like overhead for a given key request to always read 24 bytes, given the defined structure of TPT table 26, this implementation is advantageous since a given request can reference either a window entry (consisting of 24 bytes) or a region entry (consisting of 8 bytes). Address translation engine 12 only knows which type of data will be resident in a

TPT location indicated by the key when the value is actually read from system memory.

Therefore, address translation 12 may always read 24 bytes during the first access to TPT 26 on behalf of a key request. Thus, all cases where a window entry is referenced are handled, and a chance is provided that the second TPT access may be eliminated all together when a region entry is referenced. If a request is determined to be outside the bounds of TPT 26, an inbound read request is not generated. These requests are instead passed to request completion logic 34 as an error by request processing logic 58.

The paragraph beginning at page 22, line 20, is amended as follows:

In both cases, the calculated protection index may be registered in inbound request logic 32 request registers, and the request indication is set. The type of this second request may always be an 8 byte protection index request. The second request is ~~[[are]]~~ treated just like requests which originated in requests providing a protection index instead of a key value. A protection index request may always generate an 8 byte read of TPT table 26 and may never cause a second request. The read completion for a protection index request may be checked for the appropriate permissions and then returned to the requester. Only the first of the three stages discussed previously may be used to complete a protection index request.

The paragraph beginning at page 24, line 4, is amended as follows:

FIG. 7 shows a diagram of a system for enhanced channel adapter performance with completion queues and an event queue according to an example embodiment of the present invention. This figure is the same as FIG. 1 except that the address translation engine 12 is not shown for simplicity, and the completion queues 110 and even queue 120 are shown in system memory as opposed to TPT table 26 (also not shown for simplicity) as shown in FIG. 1. The dotted lines between completion queue 110 and the data buffers and work queues 24 denote that work has been completed and status has been stored in completion queue 110. A single entry in event queue 120 represents one entire completion queue. The one completion queue may contain a number of entries which indicate data buffers read or written by channel adapter 10 (i.e., work completed). Completion queue 14 off loads handling of all completion queue and event activity

from packet processing engine 16. Packet processing engine 16 communicates with host interface 18 and link interface 20 to move data from system memory to the link interface, and vice versa.

The paragraph beginning at page 25, line 7, is amended as follows:

FIG. 8 shows a block diagram of a completion engine according to an example embodiment of the present invention. Completion queue engine 14 consists of a finite state machine 132 ~~[[32]]~~ local bus registers 130, completion queue (CQ) context memory 134, event queue (EQ) work registers 136 and completion queue (CQ) work registers 138. Completion queue engine 14 communicates with local bus 22 allowing software to program completion queue registers 138 ~~[[130]]~~ in channel adapter 10. Local bus 22 allows software to program information related to the configuration of the completion queues and event queues 120 in system memory. Local bus 22 may propagate accesses from software into 32 bit reads or writes of individual registers within channel adapter 10. Completion queue engine 14 also communicates with packet processing engine 16 allowing completion and event requests to be handled. Completion queue engine 14 uses host interface 18 to write to completion queues and the event queue in system memory. Address translation engine 12, as noted previously, allows completion queue engine 14 (i.e., a requestor) to convert virtual addresses to physical addresses. Completion queue engine 14 communicates ~~communications~~ with link interface 20 allowing link events (events occurring on the switched fabric network) to be posted in event queue 120.

The paragraph beginning at page 33, line 6, is amended as follows:

FIG. 14 is a flowchart of CQ request processing according to an example embodiment of the present invention. A finite state machine receives a CQ request from a packet processing engine (PPE) S1. A busy signal is generated to the packet processing engine S2. The context memory location associated with the completion queue number is read S3. The data is loaded into the working registers S4. It is determined whether the completion queue has been enabled via the local bus S5. If the completion queue has been enabled, and if the physical page address stored in context memory 134 is invalid, an address translation is issued S6. If the completion queue has not been enabled, an error signal is generated S8. The valid/invalid indication refers to

the copy of the physical address stored in context memory 134. A translation request to the TPT may only be needed if the context memory does not contain a valid address (i.e., physical address is invalid). Finite state machine 132 may write back the physical address status as invalid when a page boundary is crossed (forcing the next access to that completion queue to request a translation, etc.).

The paragraph beginning at page 33, line 20, is amended as follows:

The completion queue engine waits for address translation to complete S6A. The address translation is then checked for errors S7, and if errors exist an error signal is generated S8. If there are no address translation errors, an inbound write request is issued to the host interface writing a new completion queue entry to the completion queue in system memory S9. It is checked to see if a solicited event was requested S10. If no solicited event was requested the process ends S18. If a solicited event was requested, it is determined ~~determine~~ whether events were enabled S11, and if so, the event is generated S14. If events were not enabled, the process ends S13. Further, if solicited events were requested S10, it is determined if there were any errors S12, and if so, the event is generated S14. If there are no errors, the process terminates S13. If the event is generated S14, the eventenable is cleared S17, and modified values are written from the work registers back into context memory S15. The busy status is then cleared S16.

The paragraph beginning at page 34, line 9, is amended as follows:

FIG. 15 shows a flowchart of an event key request processing according to an example embodiment of the present invention. A finite state machine receives an event queue request from a packet processing engine S20. A busy signal is generated to the packet processing engine S21. It is determined whether the event queue has been enabled via the local bus S22. If the event queue has been enabled, and if the physical page address stored in context memory 134 is invalid, an address translation is issued S23. If the event queue has not been enabled, an error signal is generated S32. The completion queue engine waits for address translation to complete S23A. It is then determined if the address translation has errors S24, and if so, an error signal is generated S25. If the address translation does not have errors, an inbound write request is issued

to the host interface writing a new event queue entry to an event queue in system memory S26. A check is made to determine if any errors have been associated with this event S27. If errors exist, an error signal is asserted S28. If no errors exist, it is then determined if interrupts are enabled S29, and if so, an interrupt is generated to host interface S30. If interrupts are not enabled, the busy status is cleared S31. If an interrupt is generated, the interrupt enable bit is then cleared S32. The busy status is then cleared S31.

The paragraph beginning at page 35, line 20, is amended as follows:

FIG. 20 shows a flowchart of manual completion queue entry processing according to an example embodiment of the present invention. A finite state machine receives a manual CQ entry from the local bus S80. The data is read from the local bus S81. The data is then loaded into the working registers S82. It is determined whether the completion queue has been enabled S83. If the completion queue has been enabled, if the physical page address stored in context memory 134 is invalid, an address translation is issued S84. If the completion queue has not been enabled, an error signal is generated S86. The completion queue engine waits for address translation to complete S84A. It is determined whether the address translation has errors S85, and if so, an error signal is generated S86. If the address translation does not have errors, an inbound write request is issued to the host interface writing a manual completion queue entry to a completion queue in system memory S87. It is then determined whether solicited or normal events have been requested S88, and if not, the process ends S96. If events have been requested, it is determined whether the events are enabled S89, and if not, the process ends S91. If events are requested, it is also determined if there are any errors S90 and if no errors, the process terminates S91. If events are enabled or there are errors, an event is generated S92. The event enable is cleared S95. Modified values of working registers are written back into context memory S93. The local bus CQ opcode field is cleared S94.